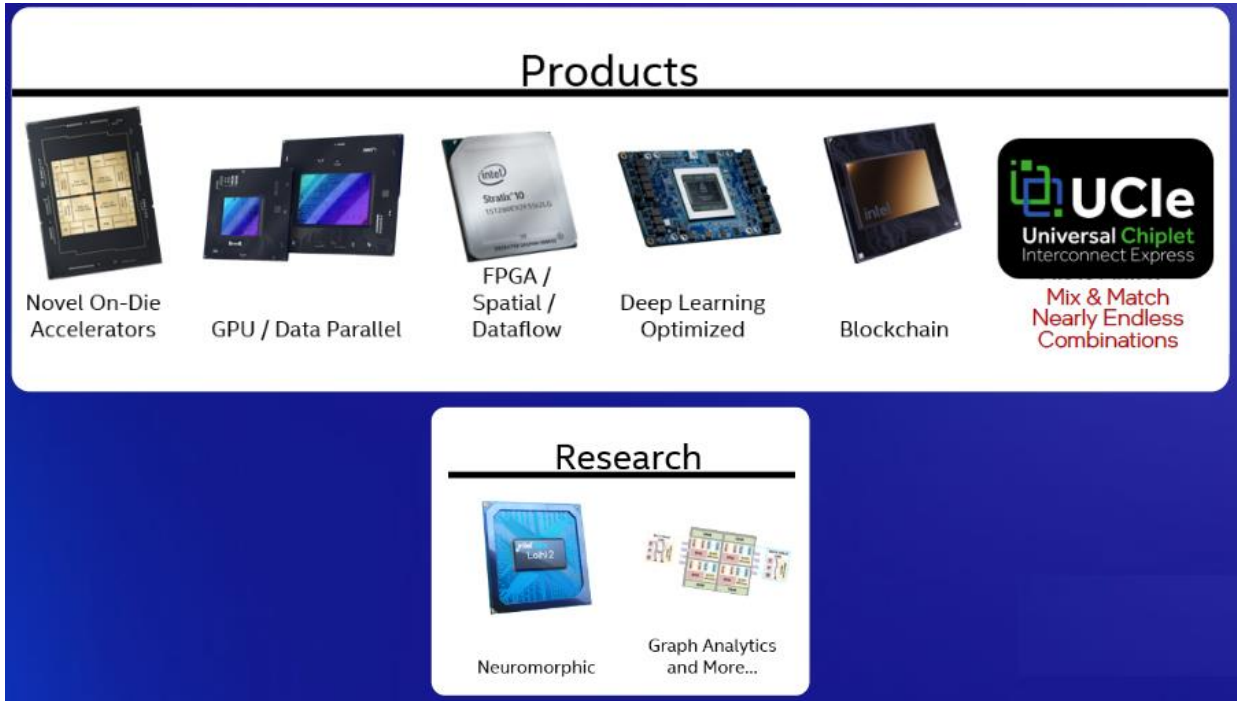
Migrating from CUDA to SYCL – Intel's one-stop portal

Chekuri S. Choudary and Rakshith Krishnappa

intel®

GPU accelerators are on the rise

- Moore’s law is ending
- Dennard scaling has ended
- GPUs are faster and power efficient
    - 98% of the Summit performance comes from GPUs
- Multiple GPU vendors
    - NVIDIA H100
    - AMD Instinct
    - Intel PVC



The next decade will see a Cambrian explosion of novel computer architectures, meaning exciting times for computer architects in academia and industry.

John Hennessy and David Patterson

A New Golden Age for Computer Architecture

CACM, Feb 2019, Vol 62, No 2, pp 48-60

Programming Challenges

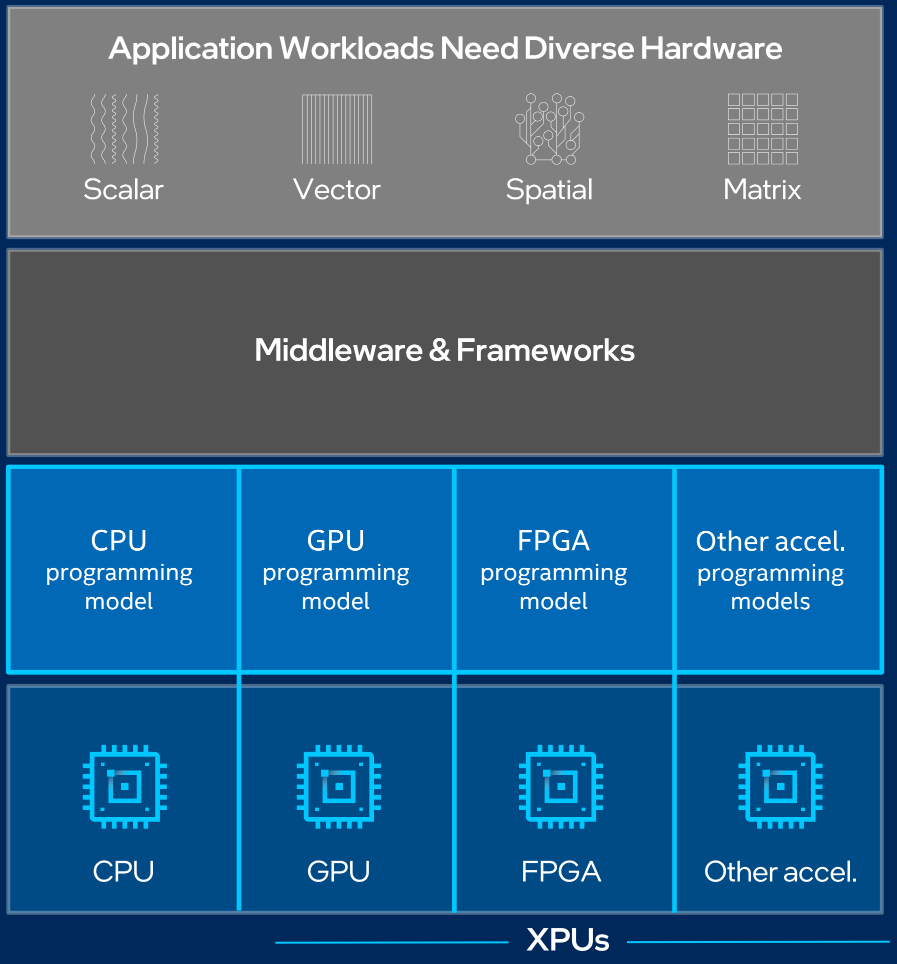
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Requires separate programming models and toolchains for each architecture

Software development complexity limits freedom of architectural choice



SYCL – Khronos standard for heterogeneous computing

Template library specification

C++ with SYCL :

- Pick a device
  - Binds a queue
- Share data
  - Unified shared memory (USM) or buffers
  - Implicit and explicit data transfers
- Offload computation
  - Submit command groups to the queue
  - Inorder and out-of-order (DAG) scheduling

CUDA to SYCL dictionary

CUDA	SYCL
Block	Work group
Thread	Work item
Grid	ND-range
Kernel	Command group
CUDA Stream	Queue
Shared memory	Local memory
Cooperative groups	Subgroups
Unified memory	Unified shared memory(USM)
Graphs	tf::syclflow in Taskflow

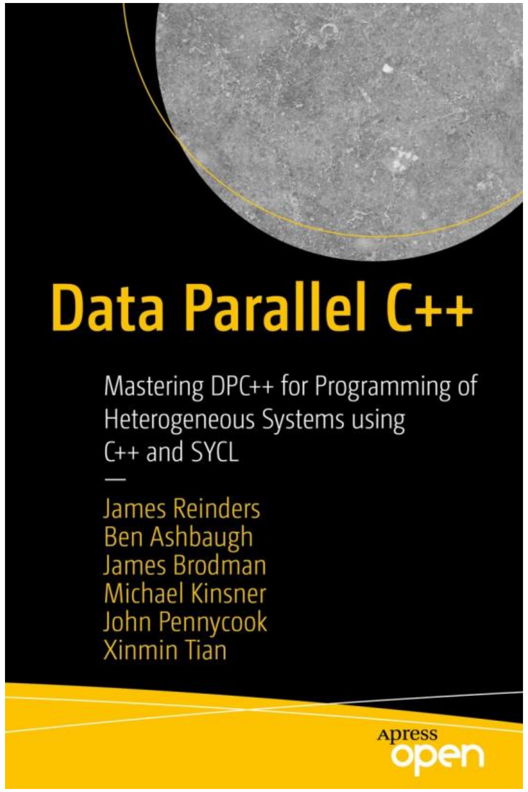
Simple SYCL program

```
#include <CL/sycl.hpp>
#include <array>
#include <iostream>
using namespace sycl;

int main() {
  constexpr int size=16;
  std::array<int, size> data;

  // Create queue on implementation-chosen default device
  queue Q;

  // Create buffer using host allocated "data" array
  buffer B { data };

  Q.submit([&](handler& h) {
    accessor A{B, h};
    h.parallel_for(size , [=](auto& idx) {
        A[idx] = idx;
        });
    });

  // Obtain access to buffer on the host
  // Will wait for device kernel to execute to generate data
  host_accessor A{B};
  for (int i = 0; i < size; i++)
    std::cout << "data[" << i << "] = " << A[i] << "\n";

  return 0;
}
```

Host code

Device code

Host code

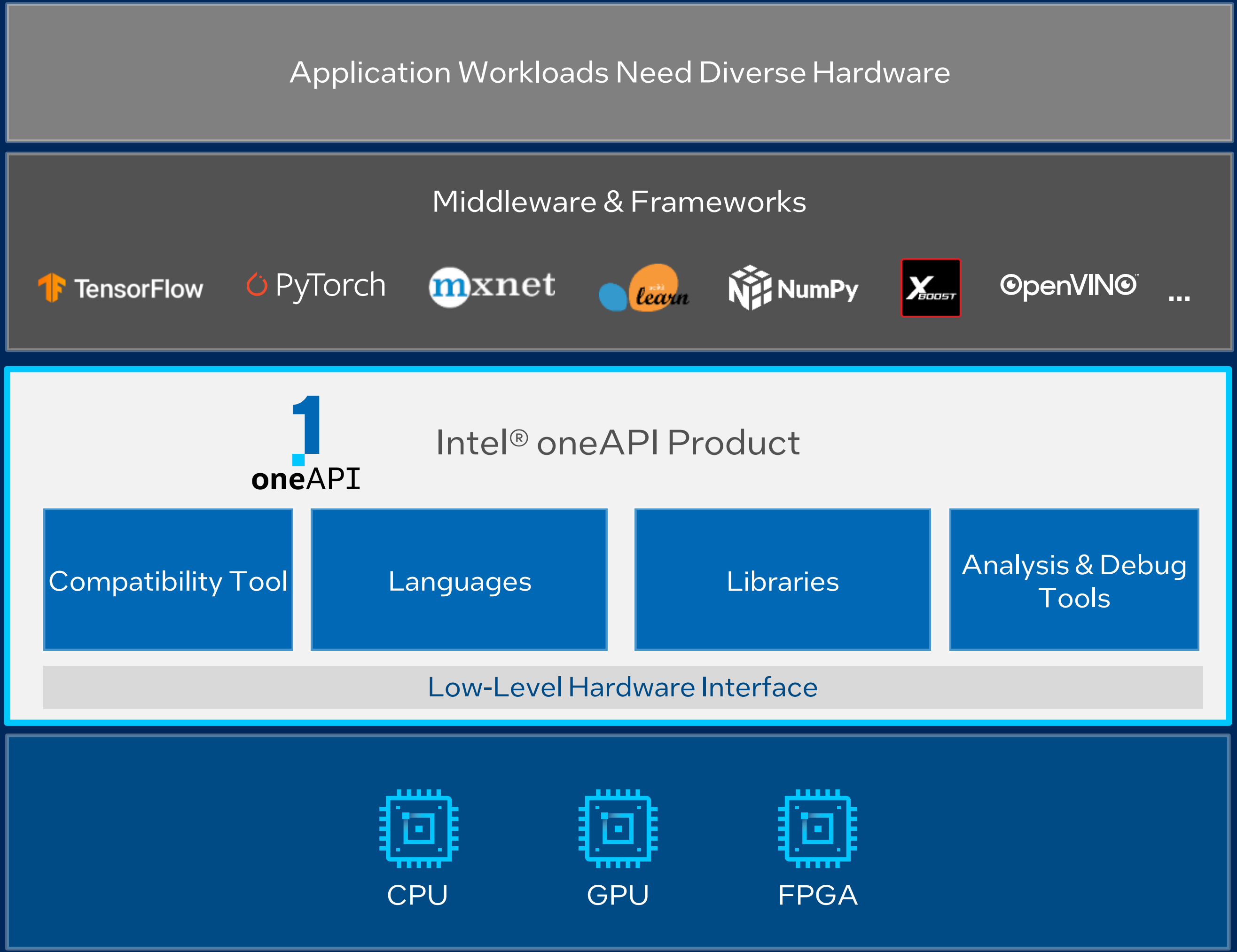


Intel® oneAPI Product

Built on Intel's Rich Heritage of CPU Tools Expanded to XPUs

A complete set of advanced compilers, libraries, and porting, analysis and debugger tools

- Accelerates compute by exploiting cutting-edge hardware features
- Interoperable with existing programming models and code bases (C++, Fortran, Python, OpenMP, etc.), developers can be confident that existing applications work seamlessly with oneAPI
- Eases transitions to new systems and accelerators - using a single code base frees developers to invest more time on innovation

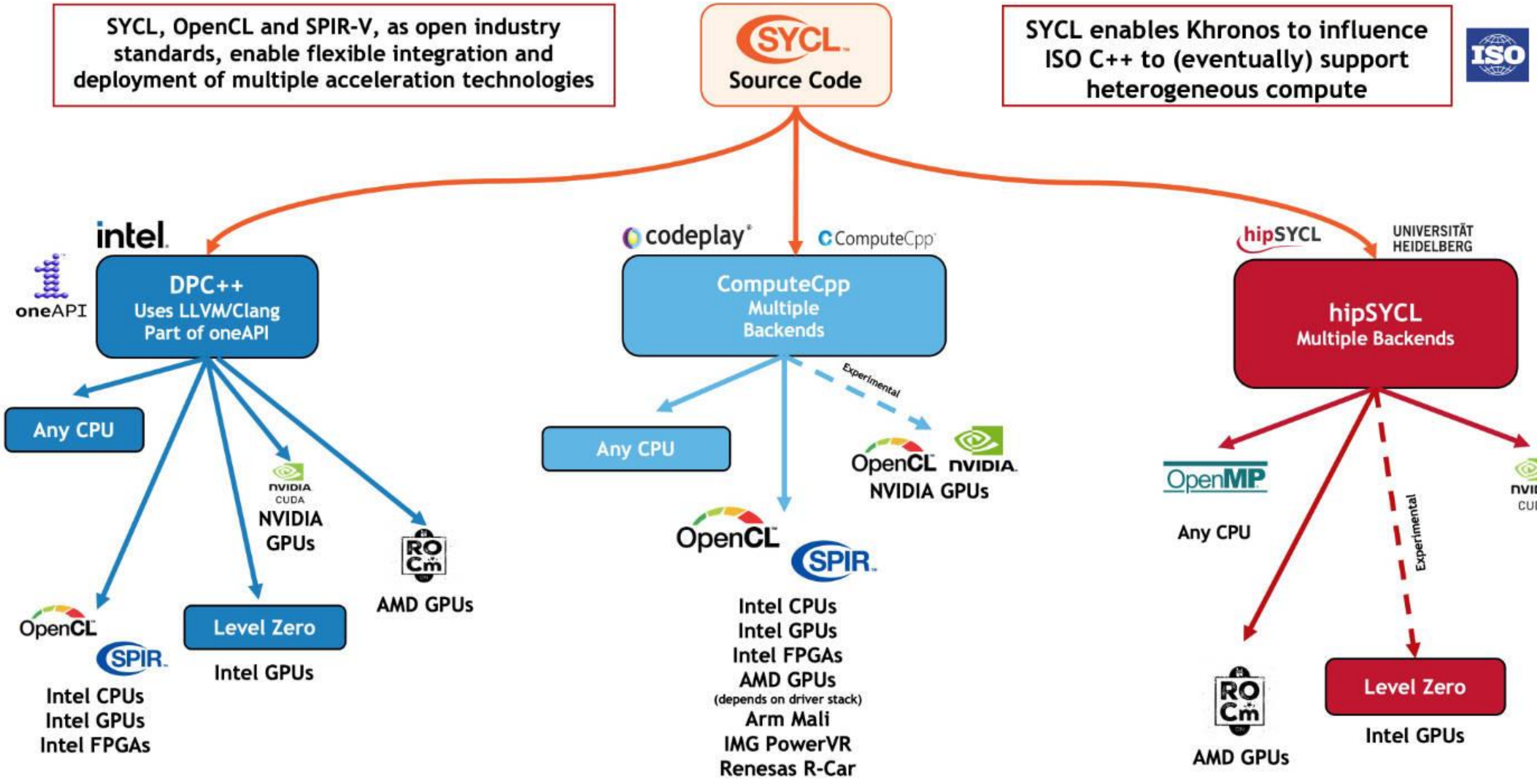


Latest version is 2023.0.0

[Available Now]

Visit software.intel.com/oneapi for more details
Some capabilities may differ per architecture and custom-tuning will still be required. Other accelerators to be supported in the future.

SYCL is gaining traction



https://www.khronos.org/sycl/

SYCLomatic migration tool

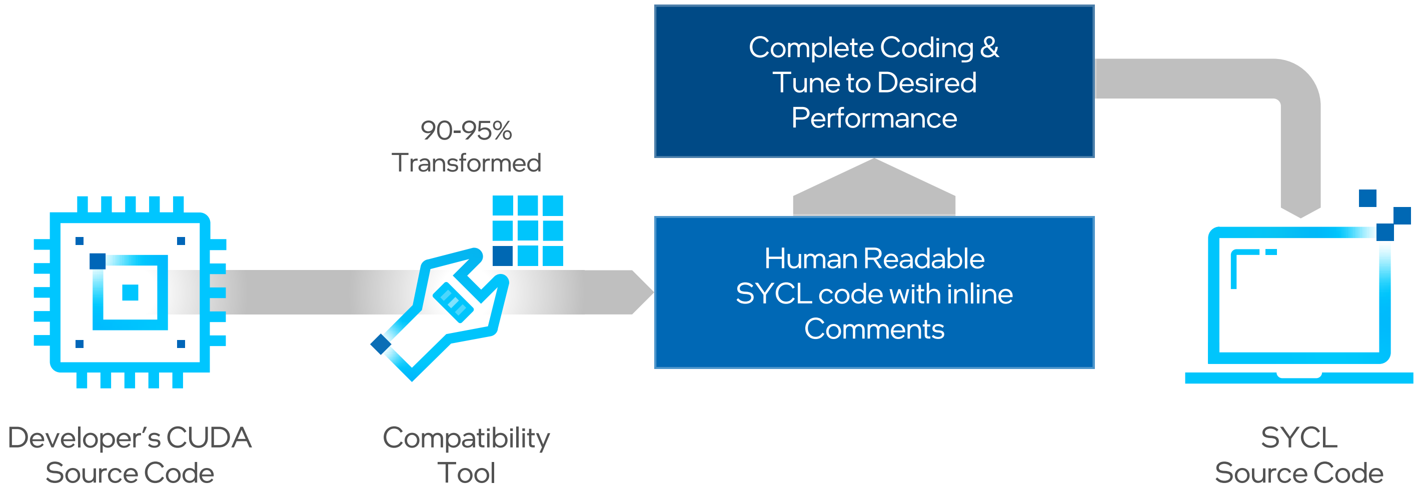
Minimizes Code Migration Time

Assists developers migrating code written in CUDA to SYCL once, generating **human readable** code wherever possible

~90-95% of code typically migrates automatically

Inline comments are provided to help developers finish porting the application

SYCLomatic Usage Flow



Vector Addition from CUDA to SYCL - Code Sample

```
#include <cuda.h>
#include <iostream>
#define N 2048

// Computation Offloaded to device
__global__ void VectorAddKernel(float* A, float* B, float* C)
{
        int id = blockDim.x * blockIdx.x + threadIdx.x;
        C[id] = A[id] + B[id];
}

int main()
{
        // Initialize data on host
        float A[N], B[N], C[N];
        for (int i = 0; i < N; i++){
                A[i] = 1;
                B[i] = 2;
                C[i] = 0;
        }

        // Allocate memory on device
        float *d_A, *d_B, *d_C;
        cudaMalloc(&d_A, N*sizeof(float));
        cudaMalloc(&d_B, N*sizeof(float));
        cudaMalloc(&d_C, N*sizeof(float));

        // Copy data from  host to device
        cudaMemcpy(d_A, A, N*sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, N*sizeof(float), cudaMemcpyHostToDevice);

        // Offload computation to device
        int nThreads = 256;
        int nBlocks = N / nThreads;
        VectorAddKernel<<<nBlocks, nThreads>>>(d_A, d_B, d_C);

        // Copy result data from device to host
        cudaMemcpy(C, d_C, N*sizeof(float), cudaMemcpyDeviceToHost);

        // Print output on host
        for (int i = 0; i < N; i++) std::cout<< C[i] << " ";
        std::cout << "\n";

        // Free device memory
        cudaFree(d_A);
        cudaFree(d_B);
        cudaFree(d_C);
        return 0;
}
```

Header file

Kernel function

Device Memory Allocation

Copy host to device

Submit Kernel Task

```
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#include <iostream>
#define N 2048

// Computation Offloaded to device
void VectorAddKernel(float* A, float* B, float* C, sycl::nd_item<3> item_ct1)
{
        int id = item_ct1.get_local_range(2) * item_ct1.get_group(2) +
                 item_ct1.get_local_id(2);
        C[id] = A[id] + B[id];
}
int main()
{
 dpct::device_ext &dev_ct1 = dpct::get_current_device();
 sycl::queue &q_ct1 = dev_ct1.default_queue();
        // Initialize data on host
        float A[N], B[N], C[N];
        for (int i = 0; i < N; i++){
                A[i] = 1;
                B[i] = 2;
                C[i] = 0;
        }

        // Allocate memory on device
        float *d_A, *d_B, *d_C;
        d_A = sycl::malloc_device<float>(N, q_ct1);
        d_B = sycl::malloc_device<float>(N, q_ct1);
        d_C = sycl::malloc_device<float>(N, q_ct1);
        // Copy data from  host to device
        q_ct1.memcpy(d_A, A, N * sizeof(float));
        q_ct1.memcpy(d_B, B, N * sizeof(float)).wait();
        // Offload computation to device
        int nThreads = 256;
        int nBlocks = N / nThreads;
        /*
        DPCT1049:0: The work-group size passed to the SYCL kernel may exceed the
        limit. To get the device limit, query info::device::max_work_group_size.
        Adjust the work-group size if needed.
        */
        q_ct1.parallel_for(sycl::nd_range<3>(sycl::range<3>(1, 1, nBlocks) *
                                             sycl::range<3>(1, 1, nThreads),
                                         sycl::range<3>(1, 1, nThreads)),
                         [=](sycl::nd_item<3> item_ct1) {
                                 VectorAddKernel(d_A, d_B, d_C, item_ct1);
                         });

        // Copy result data from device to host
        q_ct1.memcpy(C, d_C, N * sizeof(float)).wait();

        // Print output on host
        for (int i = 0; i < N; i++) std::cout<< C[i] << " ";
        std::cout << "\n";

        // Free device memory
        sycl::free(d_A, q_ct1);
        sycl::free(d_B, q_ct1);
        sycl::free(d_C, q_ct1);
        return 0;
}
```

Nsight systems (nsys) results

CUDA

CUDA API Statistics:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
82.7	140,316,263	1	140,316,263.0	140,316,263.0	140,316,263	140,316,263	0.0	cudaStreamCreate
5.9	10,078,523	1	10,078,523.0	10,078,523.0	10,078,523	10,078,523	0.0	cudaEventSynchronize
5.9	10,078,436	1	10,078,436.0	10,078,436.0	10,078,436	10,078,436	0.0	cudaEventSynchronize
1.8	3,000,369	1	3,000,369.0	3,000,369.0	3,000,369	3,000,369	0.0	cudaDeviceSynchronize
1.8	3,000,278	1	3,000,278.0	3,000,278.0	3,000,278	3,000,278	0.0	cudaDeviceSynchronize
0.5	776,707	1	776,707.0	776,707.0	776,707	776,707	0.0	cudaMallocHost
0.5	776,362	1	776,362.0	776,362.0	776,362	776,362	0.0	cudaMallocHost
0.2	336,650	2	168,325.0	168,325.0	14,016	322,634	218,225.9	cudaFreeHost
0.2	336,286	2	168,143.0	168,143.0	13,876	322,410	218,166.5	cudaFreeHost
0.1	227,590	3	75,863.3	104,461.0	7,994	115,135	59,018.4	cudaMalloc
0.1	227,035	3	75,678.3	104,309.0	7,698	115,028	59,116.1	cudaMalloc
0.1	211,097	3	70,365.7	64,750.0	7,269	139,078	66,083.7	cudaFree
0.1	210,655	3	70,218.3	64,651.0	7,111	138,893	66,067.2	cudaFree
0.0	33,574	1	33,574.0	33,574.0	33,574	33,574	0.0	cudaMemset
0.0	33,461	1	33,461.0	33,461.0	33,461	33,461	0.0	cudaMemset
0.0	23,501	2	11,750.5	11,750.5	10,018	13,483	2,450.1	cudaLaunchKernel
0.0	23,308	2	11,654.0	11,654.0	9,923	13,385	2,448.0	cudaLaunchKernel
0.0	22,164	2	11,082.0	11,082.0	6,665	15,499	6,246.6	cudaMemcpyAsync
0.0	21,997	2	10,998.5	10,998.5	6,580	15,417	6,248.7	cudaMemcpyAsync
0.0	6,864	1	6,864.0	6,864.0	6,864	6,864	0.0	cudaStreamDestroy
0.0	5,671	1	5,671.0	5,671.0	5,671	5,671	0.0	cudaHostAlloc
0.0	5,547	1	5,547.0	5,547.0	5,547	5,547	0.0	cudaEventRecord
0.0	5,518	1	5,518.0	5,518.0	5,518	5,518	0.0	cudaHostAlloc
0.0	5,438	1	5,438.0	5,438.0	5,438	5,438	0.0	cudaEventRecord
0.0	2,474	1	2,474.0	2,474.0	2,474	2,474	0.0	cudaEventCreate
0.0	1,492	1	1,492.0	1,492.0	1,492	1,492	0.0	cudaEventDestroy
0.0	1,445	1	1,445.0	1,445.0	1,445	1,445	0.0	cuModuleGetLoadingMode

SYCL

CUDA API Statistics:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
63.5	123,293,808	1	123,293,808.0	123,293,808.0	123,293,808	123,293,808	0.0	cuCtxCreate_v2
35.3	68,547,019	2	34,273,509.5	34,273,509.5	40,974	68,506,045	48,412,116.0	cuEventSynchronize
0.4	831,836	2	415,918.0	415,918.0	3,710	828,126	582,950.1	cuMemAllocHost_v2
0.2	437,493	1	437,493.0	437,493.0	437,493	437,493	0.0	cuModuleLoadDataEx
0.2	344,354	2	172,177.0	172,177.0	7,360	336,994	233,086.4	cuMemFreeHost
0.1	280,163	4	70,040.8	2,070.0	1,501	274,522	136,321.4	cuStreamSynchronize
0.1	221,702	3	73,900.7	107,512.0	3,681	110,509	60,830.5	cuMemAlloc_v2
0.1	215,362	3	71,787.3	67,129.0	6,118	142,115	68,118.1	cuMemFree_v2
0.0	23,693	4	5,923.3	3,723.5	1,941	14,305	5,770.3	cuStreamCreate
0.0	22,857	1	22,857.0	22,857.0	22,857	22,857	0.0	cuMemsetD8Async
0.0	22,299	2	11,149.5	11,149.5	7,919	14,380	4,568.6	cuLaunchKernel
0.0	21,338	2	10,669.0	10,669.0	6,902	14,436	5,327.3	cuMemcpyAsync
0.0	10,912	3	3,637.3	3,278.0	2,681	4,953	1,177.9	cuStreamDestroy_v2
0.0	9,893	7	1,413.3	519.0	348	5,627	1,894.1	cuEventRecord
0.0	7,930	7	1,132.9	493.0	274	4,723	1,600.9	cuEventCreate
0.0	1,555	5	311.0	271.0	197	475	118.7	cuEventDestroy_v2

[5/7] Executing 'gpukernsum' stats report

CUDA Kernel Statistics:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
99.6	68,532,846	1	68,532,846.0	68,532,846.0	68,532,846	68,532,846	0.0	Typeinfo name for MonteCar

https://developer.intel.com/cuda2sycl

- A one stop shop portal with all that's needed for migrating to SYCL
- High quality & deep content which showcases code samples & best practices
- Forum Support from the community including Intel engineers
- Quality examples that are inspirational – the art of possible
- Tutorials to be added

Workshop Agenda

- Feb 15: Introduction to Using the SYCLomatic Tool and Compiling/Executing SYCL code on Intel Dev Cloud
- March 15: Migrating more complex CUDA source with the SYCLomatic Tool
- April 12: Mini Hackathon: Migrating your CUDA Code to SYCL - tips, tricks, and limitations

Session #1 - 02/15/2023, 1:30 – 3:30PM CT

- **Introduction to Using the SYCLomatic Tool and Compiling/Executing SYCL code on Intel Dev Cloud**
  - Installing SYCLomatic tool
  - Understand SYCLomatic tool usage and command line options
  - Migrate a simple CUDA example with just one source file to SYCL
  - Migrate a CUDA example with multiple CUDA source files to SYCL
- In this session we will mainly try to understand how memory allocation and memory copy is accomplished in CUDA versus SYCL, we will also look at how a kernel is offloaded to run on GPU in CUDA versus SYCL.

Session #2 - 03/15/2023, 1:30 – 3:30PM CT

- **Migrating more complex CUDA source with the SYCLomatic Tool**
  - Migrate a CUDA example with multiple CUDA source files to SYCL
  - Optimize Kernel code with SYCL features.
- In this session we will understand how CUDA features like Local Memory, Cooperative groups, warp primitives and atomic operations are migrated to SYCL, we will inspect the CUDA and SYCL source and understand how migration was accomplished using SYCLomatic tool. We will also try to manually optimize the migrated SYCL code for performance using SYCL features.

Session #3 - 04/12/2023, 1:30 – 3:30PM CT

- **Mini Hackathon: Migrating your CUDA Code to SYCL - tips, tricks, and limitations**
  - This session will be a mini hackathon where you can bring your own CUDA source and try to migrate to SYCL, Intel experts will help and answer any questions you may have about the migration process.
  - We will also give an overview of how migration is accomplished when CUDA source use a library like cuBLAS or cuFFT, we will show case other CUDA to SYCL migration projects that are completed and can be used as reference. We will also learn about the current limitation of the SYCLomatic tool, we will learn about some tips and tricks when migrating CUDA to SYCL using SYCLomatic tool.

Pre-requisites

- These sessions involve 2 steps:
  - Migrating the CUDA source on CUDA development machine
  - Executing migrated SYCL source on Intel CPUs/GPUs on Intel Developer Cloud
- The audience is expected to have a CUDA development machine ready for this workshop, we will install SYCLomatic tool on the CUDA development and then migrate the CUDA source to SYCL.
- Once the code migration is complete, we will transfer the migrated SYCL source to Intel Developer Cloud to compile, execute and optimize on Intel CPUs/GPUs.
- If you do not have a CUDA development machine available, you can just watch the demonstration of step one, CUDA to SYCL migration and then do the step two on Intel Developer Cloud.

intel®